Internal Revenue Service

Java Coding Standards

Version 1.1.6
May 18, 2012

Revision History

Change Record			
Version Number	**Description of Change**	**Change Effective Date**	**Change Entered By**
1.1.1	Initial creation.	09/19/2011	Srinivas Mokkarala
1.1.2	Title change, minor formatting changes	10/06/2011	Srinivas Mokkarala
1.1.3	Minor modifications in format	10/14/2011	Srinivas Mokkarala
1.1.4	Incorporated review feedback from 10/20/11	10/21/2011	Srinivas Mokkarala
1.1.5	Updates 2.0, 4.0 and other minor changes	10/31/2011	Srinivas Mokkarala
1.1.6	Added 3.2.1, 3.2.5 and 4.6	05/18/2012	Srinivas Mokkarala

CII Number	**Change Effective Date**	**Change Entered By**

Table of Contents

List of Figures / Tables

1.0 Introduction

The purpose of this document is to provide Java coding standards to new Java developers. Standards help to create good coding practices and to develop code that is more readable. Adherence to coding standards makes it easier to maintain and enhance the functionality of the application for a long time. Most code exists for a long time. Good coding practices are essential to extend this longevity.

This document is expected to serve as a quick reference, and complements more formal standards that are available in IRMs. Some examples have been reused from the RGSS Java Standards Guide 1.1. URLs to external web sites have also been included generously throughout the document, to avoid restating well document concepts, examples and tutorials. This document is not a Java tutorial and neither is it a replacement for a tutorial or learning material.

The document provides a starting point for secure coding, basic coding standards and a few best practices.

2.0 Secure Coding

It is important to develop code that fulfils functional requirements effectively and efficiently. It is very important to eliminate, avoid or minimize security vulnerabilities from code. This section provides information about approved code-scanning tools and other references to develop secure code.

2.1 Scanning Tools

Developers must use approved scanning tools that quickly identify and report coding vulnerabilities. These tools easily integrate into the development environment. Further information about a specific tool and installation instructions etc. are available under the MITS/AD/Application Security web page at: http://adweb.irs.gov/domainareas/programmanagement/adpmosecurity/appsecurity/default.aspx. Developers are encouraged to contact the Security group within MIDS/AD for further information.

The EUES[1] web page is another good resource, and contains more exhaustive product version details at: http://coe.enterprise.irs.gov/softwareOrdering/product/listproducts.asp?avail=B&sort=all&manu=&prod=appscan&type=0&cat=0&subcat=0. The COE[2] contains a list of products with a user friendly search interface at: http://coe.enterprise.irs.gov/softwareOrdering/product/listProducts.asp. Additional details can also be viewed at the EA[3] Standards Profile web page located at: http://ea.web.irs.gov/esp/default.aspx.

2.2 Additional Information

IRM 10.8.6 provides coding standards for secure coding. The document may be viewed at: http://irm.web.irs.gov/Part10/Chapter8/Section6/IRM10.8.6.asp.

It is highly recommended to read secure coding guidelines and examples listed in the links below:

- The Open Web Application Security Project(OWASP) development guide at: https://www.owasp.org/index.php/Category:OWASP_Guide_Project
- OWASP coding examples at: https://www.owasp.org/index.php/Category:Vulnerability
- The CERT Oracle Secure Coding Standard for Java from the Software Engineering Institute, Carneige Mellon University at: https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java

2.3 Specific Examples

The links below provide very specific coding examples from OWASP, regarding SQL injection and cross site scripting:

- SQL Injection: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
- Cross site scripting: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

[1] End Users Equipment & Services

[2] Common Operating Environment

[3] Enterprise Architecture

3.0 Coding Standards

Each sub-section below describes a specific rule. Simple examples are also listed wherever possible. The following general standards hold good throughout the programming and design process:

- Secure coding standards must supersede other rules.
- The readability and understanding of the code overrides most rules in general.
- Code must be modular.
- Complex functionality in large-sized methods should be broken down into a hierarchy of methods.
- A Method should be limited to 50 lines or less, if possible.
- A lengthy line of code should be broken up or wrapped for better readability.
- Each Java source file should ideally contain a single public class or interface.[4]
- Method and variable names should be self explanatory to the extent feasible.
- Comments should be succinct.
- Override the hashCode() and equals() methods appropriately when needed. This article provides additional guidance: http://www.ibm.com/developerworks/java/library/j-jtp05273/index.html.

3.1 Declarations

Each class file will usually contain one public class or interface. The following order is recommended for all declarations in a class:

- Package name.
- Imports declaration.
- Class or Interface declaration.
- Static variables in the following order public, protected, private and default.
- Instance variables in the following order public, protected, private and default.
- Methods should be grouped according to their access: public, protected, private and default.
- Include a default constructor even if it doesn't implement anything(some frameworks require this).

Table 2 – Declaration order example

STANDARD	NON-STANDARD

`package gov.irs.eservices;`
 	
`import java.io.BufferedReader;`
`import java.io.IOException;`
 	
`/** class java doc */`
`public class HotBeverage()`
`{`
`  final int MAX_STRING_LENGTH = 256;`
`  private int totalAnswers;`
 	
`  /** javadoc info */`
`  public HotBeverage()`
`  {`
 	
`  }`
 	
`  /** Javadoc info */`
`  private getUserInput()`
`  {`
`  }`
`}`
	`package gov.irs.eservices;`
	
	`import java.io.*;`
	
	`public class HotBeverage()`
	`{`
	`  final int MAX_STRING_LENGTH = 256;`
	`  private int totalanswers;`
	`  public String answer;`
	
	`  private getUserInput()`
	`  {`
	
	`  }`
	`  public HotBeverage()`
	`  {`
	
	`  }`
	`}`

[4] Reference IRM 2.5.3.6.3.2 Java Source Files

It is recommended to explicitly import each class. For example, instead of writing “import java.io.*”, use “import java.io.IOException” etc. to import each individual class that is necessary. Integrated development environments including Eclipse make this easy for the developer.

Table 3 - Imports example

STANDARD	NON-STANDARD

`import java.awt.Color;`
`import java.swt.Graphics;`
`import java.io.BufferedReader;`
`import java.io.IOException;`
 	
`import org.junit.After;`
`import org.junit.Before;`
`import org.junit.Test;`
	`import javax.swing.*;`
	
	`import java.io.*;`
	`import org.junit.*;`
	`import org.junit.Assert.*;`
	`import java.awt.*;`

3.2 Naming Conventions

It is recommended to always use meaningful names for classes, methods and variables. Names should be verbose enough to help the reader relate to the business process or to clearly understand what process is being performed by a method or what state information is held in a variable. Note that relational databases often restrict column width to 30 characters. Class variables and instance variables that get persisted must conform to this constraint.

3.2.1 Package Names

Package name may consist of a single word or it can be hierarchical, with multiple words that are separated by periods. Lower case names are recommended to avoid conflicts and confusion with class and interface names. Begin package name with the top level domain such as: “`gov.irs.c2.bnc.du.engine;`” File sub directory names at deployment are determined according to the package names inside the interface and classes. This link provides additional information and guidance to define package names: http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html.

3.2.2 Class Names

Class names should be nouns and readers should be able to relate to the underlying business process. Class names should be in mixed case starting with a capital letter.

This link to Sun’s Java Coding Standards provides additional information with examples that may be useful: http://www.oracle.com/technetwork/java/codeconventions-141855.html#1852

Table 4 - Class naming examples

STANDARD	NON-STANDARD

`Class Employee implements Person, Human`
`{`
`}`
 	
`class CheckingAccount extends Banks`
`{`
`}`
	`class employee`
	`{`
	`}`
	
	`class logErrorCapturing extends Exception`
	`{`
	`};`

3.2.3 Method Names

Method names should start with a verb and must begin with a lower case word. For example: "computeProfitForYear." Abbreviations and acronyms should not be in uppercase. Ensure that the method declaration is readable and fits into the line. When a method declaration is very long, split the declaration into multiple lines, but aligning the input arguments, for clarity and readability.

Table 5 - Method naming examples

STANDARD	NON-STANDARD

`public String getDvdName();`
`computeTotal();`
`computeTotalSale();`
	`public String getDVDDName ( );`
	`ComputeTotal();`
	`computeTotalSale ();`

Keep method names verbose to the extent that the method can convey the business process unambiguously.

STANDARD	NON-STANDARD

`computeAverageTaxReturn();`
`compareTotalIncrease();`
`findHomeLoanDefaulter();`
	`computeAv();`
	`cmpTtls();`
	`findNm();`

3.2.3.1 Common Accessor Methods

These are methods that retrieve an object or primitive data type. It is recommended to prefix these methods with "get" or "set" depending on whether retrieval or update is being performed. This style keeps method names intuitive, follows parallelism in naming convention and makes it easy to read. Example method names are getLeaveBalance(), setLeaveBalance() and so on. Always choose simple and direct names to clearly convey the underlying purpose of the method.

Table 6 – Accessor method examples

STANDARD	NON-STANDARD

`getLastName();`
`setLastName();`
 	
`getEmployee();`
`setEmployee(Employee employee);`
	`retrieveLastName();`
	`stLastName();`
	
	`gtEmployee();`
	`stEmployee(Employee employee);`

3.2.3.2 Methods returning boolean

It is recommended that methods returning a boolean variable should prefix method name with "is" for example "isCreditRatinkOk()" etc.

Table 8 – Java boolean method examples

STANDARD	NON-STANDARD

`isTankFull();`
`boolean check = hasPennies();`
`isProcessComplete();`
	`tankFull();`
	`anyPennies();`
	`processCheck();`

3.2.4 Variables

Use mixed case naming for variables. Choose names that provide clarity to a reader. A variable name is in mixed case format starting with lower case letter. The following table lists a few recommendations.

Table 9 - Variable naming Syntax examples

STANDARD	NON-STANDARD

`int customerNum;`
`long accountNo;`
`Employee eopsEngineer;`
 	
// the following are clearer
int customerNum;
long accountNo;
Employee eopsEngineer;
 	
// Prefix "I" for iterator names
`Iterator iCustomer;`
Iterator iEmployee;
	`int CustomerN;`
	`long AcctNo;`
	`Employee emp;`
	
	int customerNo;
	long AcctN;
	Employee emp;
	
	`Iterator customer;`
	Iterator employee;

3.2.4.1 Constants

Use upper case names to denote constants or final variables, and separate whole words with underscore character.

Table 11 - Final variable examples

STANDARD	NON-STANDARD
`final static int MAX_PATH_LENGTH = 200;` `final static int COLOR_BLUE = 10;`	`Final static int maxPathLength = 200;` `final static int COLORBLUE = 10;`

Enumerators or "enums" is recommended whenever it is appropriate. **Additional information** with examples is available at: http://download.oracle.com/javase/1,5.0/docs/guide/language/enums.html.

3.2.4.2 Use named constants.

Avoid hard-coded numeric and literals inside code. Define appropriate constants, and use the constant names inside code blocks.

Table 12 – Named constants

STANDARD	NON-STANDARD

`static final float PI = 3.14f;`
 	
`public float computeCircumference(float radius)`
`{`
`  return 2 * PI * radius;`
`}`
	`public float computeCircumference(float radius)`
	`{`
	`  return 2 * 3.14f * radius;`
	`}`

3.2.4.3 Strings

Java String objects are not mutable, and can potentially use lot of memory space. It is recommended to use StringBuffer for large Strings since this is mutable, and can be reused. Another alternative is StringBuilder, but be aware that class is not thread safe.

Java String class provides additional optimization for applications using very high volume of Strings. The String.intern() method is useful in such scenarios. It explicitly instructs the JVM to maintain a single copy of the String. Additional explanation can be found in API documentation and also at: http://en.wikipedia.org/wiki/String_interning. Most smaller applications will not gain much performance.

3.2.4.4 Externalizing Strings

Sometimes an application may require to define and use very high volume of string constants and messages. In such cases it is recommended to externalize those strings, by storing them in property files and using Java's ResourceBundle and MessageFormat classes to retrieve and format them. This allows better management and documentation of the application messages and would also provide the ability to change a message without being forced to modify the class or classes using it.

3.2.5 Java Archives

It is recommended to use lower case names for jar, war and ear files, to the extent it is possible.

3.3 Statements

3.3.1 Initialization

Java initializes variables with well defined default values. Section 4.5.5 from this link provides a good overview: http://java.sun.com/docs/books/jls/second_edition/html/typesValues.doc.html. It is recommended to explicitly indicate initial values, to err on the side of caution, given the multitude of computing environments.

Table 13 - Initialization statement examples

STANDARD	NON-STANDARD

`int nCars = 0;`
`double valueCars = 0.0L;  // L suffix`
`String myString = null;`
`TaxPayer taxPayer = null; // Object`
	`int nCars;`
	`Double valueCars;`
	`String myString, myText;`
	`TaxPayer taxpayer;`

3.3.2 Typecasting

Use explicit typecast to clearly show underlying data conversion. Whenever possible analyze and correct the code to use similar data types. Performance may be reduced because of the data conversion time lost during autoboxing and unboxing of objects.

The following URL provides relevant documentation about conversions and promotions: http://java.sun.com/docs/books/jls/third_edition/html/conversions.html

Table 14 Typecasting examples

STANDARD	NON-STANDARD

`int carsSold = 30;`
`double myValue;`
`long carsInventory = 50000;`
 	
`myValue = (double)carsSold;`
	`int carsSold = 30;`
	`double myValue;`
	`long carsInventory = 50000;`
	
	`myValue = carsSold;`

3.4 Alignment

Integrated development environment(IDE) allows developers to set tab stops, and also generates training braces etc. Use this to ensure that code has alignment that is consistent, and is readable. Be aware that different environments such as “vi” or “emacs” or “Wordpad” can have different alignment settings. Uniformity must be ensured, at least within a project. Ensure that the code contains a single statement, assignment or assignment per line. Ensure that code is aligned properly and contains adequate indentation for visual clarity.

Table 15 - Alignment examples

STANDARD	NON-STANDARD

`String myUser, // stores user name`
`myText; // stores something else`
`i++;`
`j--;`
`k++;`
`a = 19;`
`b = a;`
	`String myString, myTexxt;`
	`i++; jj--; k++;`
	`a = b = 19;`

3.4.1 Line Length

There is no explicit coding standard or limit. Long lines should be wrapped around,for readability. Here are some examples: http://www.oracle.com/technetwork/java/codeconventions-136091.html#313

3.4.2 Method Declaration

It is recommended to leave a space character before and after the last method argument. Use a space after the comma character, to separate each method argument.

Table 16 - Method declaration example

STANDARD	NON-STANDARD

`public void addItem( String name, String description, double price )`
`{`
`// See blank after "(" and before ")"`
`}`
 	
`public LoggingInfo(String user, String pwd )`
`{`
`// do something`
`}`
	`public void addItem(String name, String description, double price)`
	`{`
	
	`}`
	
	`public LoggingInfo(String user, String pwd)`
	`{`
	`}`

3.4.3 Loops and Conditionals

The following are recommended for loops, if-else and switch statements:

Always use "{" before the first statement and "}" after the last statement in loops and conditionals.
A separate method or function is recommended if more than four levels of conditional statements must be coded.

Table 17 - Loops and conditionals example

STANDARD	NON-STANDARD

`if (condition)`
`{`
`  One line statement`
`}`
`else`
`{`
`  if (condition)`
`  {`
`  statements(s)`
`  }`
`}`
 	
`for (loop control expressions)`
`{`
`  statements(s)`
`}`
 	
`while (condition)`
`{`
`  statements(s)`
`}`
	`if (condition)`
	` One line statement`
	`else {`
	`  if (condition)  {`
	`  statements(s)`
	`  for (loop) {`
	`  if (true)`
	`  {  while(loop) {`
	`  }`
	`  }`
	`  }`
	`  }`
	`}`
	
	`for (loop control expressions) {`
	`  statements(s)`
	`}`
	
	`while (condition) {`
	`  statements(s)`
	`}`

This link provides additional examples: http://incubator.apache.org/ace/java-coding-style-guide.html

3.5 Exceptions

User defined exception classes are named with the key word 'Exception' at the end. A simple scenario is shown in the table below. Here are some useful tips:

Log any known and relevant information that will be useful to (a) determine the root cause (b) recover from the exception. Ensure to release resources such as database Connection/Statement/file-handles etc. as required inside a "finally" block. This will ensure predictable resource usage and release each time the method executes. Sometimes a method may want to bubble the Exception up to the calling program, so that the caller may handle it. This can be done by re-throwing the Exception that was caught.

This Java tutorial may be useful: http://download.oracle.com/javase/tutorial/essential/exceptions/

Table 18 – Defining custom exception by extending java.lang.Exception example

STANDARD	NON-STANDARD

`Class AccessException extends Exception`
`{`
`}`
	`class AccessErrorHandling extends`
	`Exception`
	`{`
	`}`

3.7 Logging

System logs reflect the status of an application. Information in log files should be succinct yet sufficient to facilitate the following:

- Checking system health to see whether any errors/exceptions exist and require investigation.
- Debugging of issues relating to interoperability, functionality.
- Determining root cause of failures.
- Verify application workflows.
- Garbage collection cycles.
- Audit trail logging.

Logging may be very informative for debugging purposes or succinct to list warnings and critical system behavior. Additional information can be viewed at: http://logging.apache.org/log4j/1.2/manual.html

3.8 Code Comments

The developer must add appropriate yet succinct comments that provide adequate explanation to readers. Java comments must start with '/*' and end with '*/' for block comments in one or more lines. Alternately '//' can be used anywhere in a line to start a comment, for single line comments.

Table 19 - Comment alignment example

STANDARD	NON-STANDARD

`While (true)`
`{`
`  if (myMoney < 0.0)`
`  {`
`  /* Insufficient funds found;`
`  Debit check bounce fee */`
`  }`
`  else`
`  {  // withdrawl`
`  }`
`}`
	`While (true)`
	`{`
	`// out of money`
	`  if (myMoney < 0.0)`
	`  {`
	`/*not making enough */`
	`  }`
	`  //sell everything`
	`  else`
	`  {`
	`  // making too much`
	`  }`
	`}`

3.9 Javadoc

Java Standard Deveopment Kit (SDK) provides a tool called "javadoc" that parses Java code for pre-specified comments tags, and generates a html page that can be published. Javadocs also picks up block comments (enclosed within '/**' and '*/'). Developers must document all public variables and public methods.

This article provides more details about Javadoc syntax, and where to include such comment tags: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html. Some of the tags used by Javadoc are shown below.

Table 20 - Javadoc tag definitions

Tag	Description	Format	When to Use
@author	Identifies the author of a class.	@author name-text	overview, package, classes and interfaces only, required
@version	Specifies the version of a class.		classes and interfaces only, required. See **footnote 1**
@param	This tag is used to define parameters that are passed into a method	@param parameter-name description	methods and constructors only
@return	This tag defines values that are returned from methods	@return description	Methods only
@exception	Identifies an exception thrown by a method.	@exception class-name description	
@see	This tag creates "See Also:" output. Usually used to refer to related classes	@see "string" @see label @see package.class#member label	

3.9.1 File Documentation

After the package declaration and before the imports, the file documentation should be provided. The file documentation describes any assumptions or warning that the programmer should be aware of and any legal descriptions that normally would not be exposed in the java docs output.
The header information at the top of the file documentation provides a legal banner. An example of the file documentation format is provided below:

Table 21 - Default legal text

```
/**
 * United States of America - Official Use Only The US Government possesses the
 * unlimited rights throughout the world for Government purposes to publish,
 * translate, reproduce, deliver, perform, and dispose of the technical data,
 * computer software, or computer firmware contained herein; and to authorize
 * others to do so.
 */
```

3.9.2 Class Documentation

After the imports statements and defined constants, the class documentation provides a description and a purpose for the class. This information is useful to a developer implementing the class, similar to Sun Java Documentation for Java. The Class documentation contains the author, java version, class version and history. (javadocs tags, @author, @version, @since, @history) The following tags are optional; @see, @serial and @deprecated depending upon the class implementation.

Here is an example of the class declaration:

Table 22 - Class declaration example

```
/**
 * Class description: <br>
 * Main Entity populated by IMFTaxModParsingFactory.<br>
 * This is the parsed object that is returned to the requesting application.<br>
```

```
 * Layout of this object mimics respective SCAP section in Excel layout <br>
 * produced by database owner.
 *
 * @author Developer Name, Internal Revenue Service
 * @since 1.5
 * @version 1.0 - 2010-02-08
 * @history 2010-02-20 DR59005 Developer Name, Initial Development
 */
```

3.9.3 Method Documentation

The method documentation provides a description and a purpose for the procedure along with the history of changes by the developer. The incoming parameters, return values and exceptions thrown are included in the documentation if implemented. (See Javadoc @param, and @return).

Table 23 - Method documentation example

STANDARD

```
/**
 * Convert LDBResponse data to XML format to be sent back to requestor. <br>
 * If the response contains errors: <br>
 *  - If a SCAP error, include SCAP error code and details of the error. <br>
 *  - If LDB error, include error details and add "Error" parameter to "TtbError" tag.
 * <br>
 * <br>Attach the "Appendix" section to the response.
 * @param ldbResponse Response object containing SCAP or Error data.
 * @param appendix "Appendix" section carried over from request to be passed back
 *                 as part of response.
 * @return XML-formatted payload
 * @history 2010-02-20 DR59005 Developer Name Initial Development
 */
public String convertResponseToXML(LDBResponse ldbResponse, String appendix)
{
}
{
```

NON-STANDARD

```
/**
*/
public String convertResponseToXML(LDBResponse ldbResponse, String appendix)
{
}
```

4.0 Best Practices

Software work products consist of science and engineering at their core, along with some art. The underlying architecture, algorithms, data structures, protocols, performance-scalability-engineering, reuse, design, interoperability etc. reflect the science and engineering aspects. User interface design and layout design generally reflect the art portion. Best practices for architecture, design, development, deployment, debugging, problem-solving including design patterns, session management, JVM/GC tuning are high value skills and are difficult to condense into short document. Design and code review, design patterns, performance and scalability, garbage collection and some general best practices are mentioned very briefly. Readers are encouraged to research further.

4.1 Design and Code Review

Continuously review software designs, code, test cases, interoperation between components and the various business processes. These questions may serve as an outline checklist:

- Does the code follow standards?
- Does the code conform to secure coding standards?
- Has code implemented functional requirements fully?
- Is the control flow consistent with business processes?
- Are input/output parameters consistent with documented API?
- Are methods and API generic (desirable) or specific? For example it is desirable to use java.util.Map instead of java.util.HashMap to make the API more generic.
- Do methods handle error-cases robustly, and is there adequate logging?
- Are resources being released when code encounters errors/exceptions?
- Does the code take steps to check references to avoid NullpointerExceptions?
- Is the problem sufficiently decomposed or does it appear to be monolithic?
- Does the design include commonly identifiable design patterns or is it proprietary?
- Can a software solution be reused instead of being coded?
- Is the data model normalized appropriately?
- Does persistency tier use proper transaction semantics for local/distributed transactions?

4.2 Design Patterns

Software development can be viewed as solving business problems that can often be decomposed into smaller problems. Solutions for many commonly occurring software problems have been documented, and can be reused. Also these solutions are agnostic of the implementation environment. Some popular design patterns can be viewed at: http://www.oracle.com/technetwork/java/catalog-137601.html and http://java.sun.com/developer/technicalArticles/J2EE/patterns/PatternsIntroduction.html#strategies.

4.3 Performance and Scalability

Application performance and scalability reflects the engineering and science in the architecture, design, implementation, choice of technology, JVM and database settings and so on. Online applications generally imply higher user expectations for throughput and responsiveness. This URL is a small start: http://download.oracle.com/docs/cd/B32110_01/web.1013/b28952/bestprac.htm#CBBBAFCJ.

4.4 Garbage Collection

These white paper and links within it provide a good perspective to the reader: http://192.9.162.55/performance/reference/whitepapers/5.0_performance.html.

4.5 General J2EE Best Practices

This is a science by itself, especially in large web-based applications that often serve millions of customers within a high-availability and 24x7x365 type of scenario. Some of the best practices for Enterprise Java Beans, Java Server Pages, and session management can be viewed at: http://download.oracle.com/docs/cd/B32110_01/web.1013/b28952/bestprac.htm#CBBBAFCJ

4.6 Additional Resources

The Java Guidance document from Enterprise Architecture has additional information about Java open source software and Java technologies etc. This document can be found on the JAAG web page at: http://ss.ds.irsnet.gov/sites/ea/ATICv1/DPEAG/Central%20Maven%20Repository%20for%20New%20JAVA%20guidance/Forms/AllItems.aspx

5.0 References

1. IRS Coding Standards “IRM2.5.3” March 1, 2007 Department of the Treasury. Referenced on 9/26/2011 at: http://irm.web.irs.gov/Part2/Chapter5/Section3/IRM2.5.3.asp
2. Sun Java Standards“Code Conventions for the Java Programming Language” by Sun Microsystems, Inc. Revised April 20, 1999. Reference: 9/26/11 http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
3. Learning The Java Language by Sun Microsystems, referenced on 9/26/2011 at: http://download.oracle.com/javase/tutorial/java/index.html
4. Java Tutorials by Sun Microsystems, referenced on 9/26/2011 at: http://download.oracle.com/javase/tutorial/
5. Apache Commons Logging by Apache Foundation, referenced on 9/25/2011 at: http://commons.apache.org/logging/
6. Coding conventions for JSP by Sun Microsystems, referenced on 9/26/2011 at: http://java.sun.com/developer/technicalArticles/javaserverpages/code_convention/
7. Blueprints, Guidelines, and code for end-to-end Java applications – Naming Conventions for Enterprise Applications Early Access 2 by Sun Microsystems(Oracle Corp.), referenced on 4/27/2012 at: http://java.sun.com/blueprints/code/namingconventions.html.

Appendix-A List of Acronyms

API	Application Programming Interface
EJB	Enterprise Java Beans
GC	Garbage Collector
JPA	Java Persistence API
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
JSP	Java Server Pages
J2EE	Java 2 Enterprise Edition
JMS	Java Messaging Service
IRM	Internal Revenue Manual
MITS	Modernization & Information Technology Services
AD	Applications Development
EUES	End Users Equipment & Services
COE	Common Operating Environment
EA	Enterprise Architecture
JAAG	Java Applications Architecture Group